

# On Modelling and Validation of the MQTT IoT Protocol for M2M Communication

Alejandro Rodriguez<sup>1</sup>,  
Lars Michael Kristensen<sup>1</sup> and Adrian Rutle<sup>1</sup>

<sup>1</sup>Western Norway University of Applied Sciences, Bergen  
{arte, lmk, aru}@hvl.no

**Abstract.** Machine to Machine (M2M) communication and Internet of Things (IoT) are becoming still more pervasive with the increase of communicating devices used in cyber-physical environments. A prominent approach to communication between distributed devices in highly dynamic IoT environments is the use of publish-subscribe protocols such as the Message Queuing Telemetry Transport (MQTT) protocol. MQTT is designed to be light-weight while still being resilient to connectivity loss, component failures, and loss of packets. We have developed a formal model of the MQTT protocol logic covering all three quality of service levels provided by MQTT (at most once, at least once, and exactly once). For the initial verification of the protocol model, we show how an incremental model checking approach can be used to reduce the effect of the state explosion problem. This is done by exploiting that the MQTT protocol operates in phases comprised of connect, subscribe, publish, unsubscribe, and disconnect.

**Keywords:** Coloured Petri Nets, Modelling, Verification, Communication Protocols, Internet of Things

## 1 Introduction

Publish-subscribe messaging systems [9] support data-centric communication and have been widely used in enterprise networks and applications, mainly due to scalability and support for dynamic application topology. The interaction and exchange of messages between clients following the publish-subscribe paradigm are always undertaken using an intermediary usually called a broker (or a bus) that manages topics. A client acting as a publisher on a given topic can send messages to other clients acting as subscribers to the topic without the need to know about the existence of the receiving clients.

A software system architecture based on publish-subscribe messaging provides more scalability than the traditional client-server architecture. One reason for this is that operations carried out in the broker can be highly parallelized and handled using event-driven techniques. The decoupling that the event service provides between clients acting as publishers or subscribers can be decomposed along the following three dimensions:

**Space decoupling:** Clients only need to be aware of the connection to the broker (bus).

**Time decoupling:** Clients can exchange messages without being executing at the same time.

**Synchronisation decoupling:** Activities on clients are not interrupted during publishing or receiving.

MQTT [3] is a publish-subscribe messaging transport protocol designed with the aim of being light-weight and easy to implement. These characteristics make it a suitable candidate for constrained environments such as Machine-to-Machine communication (M2M) and Internet of Things (IoT) contexts where a small memory footprint is required and where network bandwidth is often a scarce resource. Even though MQTT has been designed to be easy to implement, it still contains relatively complex protocol logic for handling connections, subscriptions, and the various quality of service levels related to message delivery. Furthermore, MQTT is expected to play a key role in future IoT applications. This means that MQTT will be implemented for a wide range of platforms and in a broad range of programming languages making interoperability a key issue. This, combined with the fact that MQTT is only backed by an (ambiguous) natural language specification, motivated us to develop a formal and executable specification of the MQTT protocol.

We have used Coloured Petri Nets (CPNs) and CPN Tools [13] for the development of the executable MQTT specification as these have been successfully applied in earlier work to build formal specifications of communication protocols [8], data networks [5], and embedded systems [1]. To ensure the proper operations of the constructed CPN model, we have validated the CPN model using simulation and verified an elaborate set of behavioural properties of the constructed model using model checking and state space exploration. In the course of our work on the MQTT specification [3] and the development of the CPN model, we have identified a number of issues related in particular to the implementation of the quality of service levels. We suspect these will be a source of interoperability problems between implementations. Compared to earlier work on modelling and verification of publish-subscribe protocols [20, 4, 10] (which we discuss in more details towards the end of this paper) our work specifically targets MQTT and we consider a more extensive set of behavioural properties.

The rest of this paper is organised as follows. In Sect. 2 we present the MQTT protocol context and give a high-level overview of the constructed CPN model. Section 3 details selected parts of the CPN model of the MQTT protocol. In Sect. 4 we present our experimental results on using simulation and model checking to validate and verify central properties of MQTT and the CPN model. Finally, in Sect. 5 we sum up the conclusions, discuss related work, and outlines directions for future work. Due to space limitation we cannot present the complete CPN model of the MQTT protocol. The constructed CPN model is available via [16]. The reader is assumed to be familiar with the basic concepts of CPNs [13].

## 2 MQTT Protocol Context and CPN Model Overview

During the last years there have been a huge increase in the use of devices related to Internet of Things (IoT) [18]. The International Union (ITU) [12] defines IoT as a global infrastructure for the information society, enabling advanced services by interconnecting physical and virtual devices. The fields of application for IoT technologies are broad, and IoT solutions are being adopted in several different environments.

At its core, IoT is characterised by the combination of physical and digital components to create new products and enable novel business models [18]. Thanks to increasingly efficient power management, broadband communication, reliable memory and advances in microprocessor technologies, it has become possible to digitalize functions and key capabilities of industrial products [19].

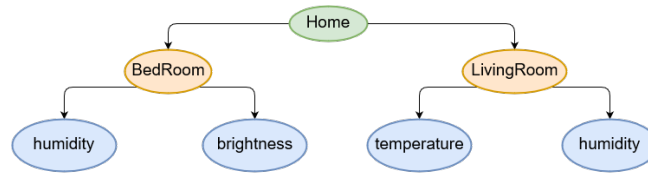
### 2.1 The MQTT Protocol

The IoT paradigm blends the virtual and the physical worlds by bringing different concepts and technical components together: pervasive networks, miniaturisation of devices, mobile communication, and new ecosystems [6]. As described in [18], from the technological side, the implementation of a connected product typically requires the combination of multiple software and hardware components distributed in a multi-layer stack of IoT technologies. A typical IoT technology stack is divided in three core layers:

- Device layer:** This layer encompasses the hardware. Additional sensors, actuators, or processors can be added to existing core hardware components, and embedded software can be modified or integrated to manage and operate the functionality of the physical device.
- Connectivity layer:** Communication protocols such as MQTT enable the communication between the individual elements and the cloud.
- Cloud layer:** This layer manages all the devices connected and the data generated by them. Developers can provide applications for the communication between the heterogeneous devices that constitute the IoT environment and derive knowledge based on data analytics.

MQTT [3] runs over TCP/IP, or over other network protocols that provide ordered, lossless and bidirectional connections. It uses the publish/subscribe message system combined with the concept of topics to provide one-to-many message distribution. The message transport structure is agnostic to the payload contained, i.e., the actual content that will be sent in the message.

*Topics.* MQTT applies topic-based filtering of messages with a topic being part of each published message. The broker uses the topics to determine whether a subscribing client should receive the message or not. Clients can subscribe to as many topics as they are interested in. A topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator).



**Fig. 1.** MQTT topic examples.

This topic structure keeps a tree hierarchy, so clients can subscribe to specific topics, levels, and groups of topics easily.

Figure 1 shows an example for a possible topic distribution for handling some sensors in a house. If a client subscribes to the topic `Home/BedRoom/temperature` it will only receive notifications from the temperature sensor in the bedroom. If the subscription is made to the topic `Home/+humidity` the receiver will get messages from the humidity sensors in the house. Here, the `+`-wildcard allows to subscribe in a single level of the topic hierarchy and as a result both the `BedRoom` and `LivingRoom` matches into the pattern. A client subscribed to the topic `Home/#` will get messages from all the devices in the house. Here, the `#`-wildcard is used to subscribe to multiple levels.

*Quality of Service.* The MQTT protocol delivers application messages according to the three Quality of Service (QoS) levels defined in [3]. The delivery protocol is symmetric, and the clients and the broker can each take the role of either a sender or a receiver. The delivery protocol is concerned solely with the delivery of an application message from a single sender to a single receiver. When the broker is delivering an application message to more than one client, each client is treated independently. The QoS level used to deliver an outbound message from the broker could differ from the QoS level designated in the inbound message. Therefore we need to distinguish two different parts of delivering a message: a client that publishes to the broker and the broker that forwards the message to the subscribing clients. The three MQTT QoS levels for message delivery are:

**At most once: (QoS level 0):** The message is delivered according to the capabilities of the underlying network. No response is sent by the receiver and no retry is performed by the sender. The message arrives at the receiver either once or not at all. An application of this QoS level is in environments where sensors are constantly sending data and it does not matter if an individual reading is lost as the next one will be published soon after.

**At least once (QoS level 1):** Where messages are assured to arrive, but duplicates can occur. It fits adequately for situations where delivery assurance is required but duplication will not cause inconsistencies. An application of this are idempotent operations on actuators, such as closing a valve or turning on a motor.

**Exactly once (QoS level 2):** Where messages are assured to arrive exactly once. This is for use when neither loss nor duplication of messages are ac-

ceptable. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

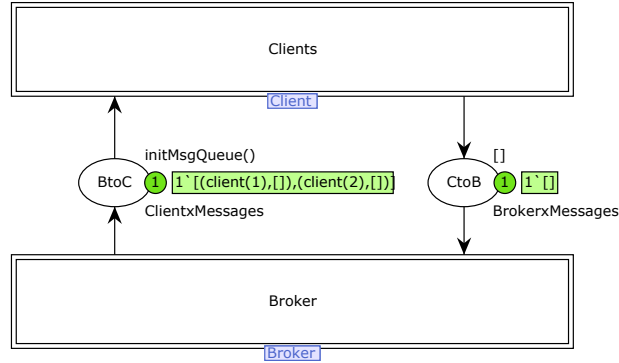
When a client subscribes to a specific topic with a certain QoS level it means that the client is determining the maximum QoS that can be expected for that topic. When the broker transfers the message to a subscribing client it uses the QoS of the subscription made by the client. Hence QoS guarantees can get downgraded for a particular receiving client if subscribed with a lower QoS. This means that if a receiver is subscribed to a topic with a QoS level 0, no matter if a sender publishes in this topic with a QoS level 2, then the receiver will proceed with its QoS level 0.

## 2.2 CPN Model Overview

Figure 2 shows the top-level module of the CPN model which consists of two substitution transitions (drawn as rectangles with double-lined borders) representing the Clients and the Broker roles of MQTT. Substitution transitions constitute the basic syntactical structuring mechanism of CPNs and each of the substitution transitions has an associated submodule that models the detailed behaviour of the clients and the broker, respectively. The name of the submodule associated with a substitution transition is written in the rectangular tag positioned next to the transition. The CPN model of the MQTT protocol consists of 24 modules organised into six hierarchical levels. We have constructed a parametric CPN model which makes it easy to change the number of clients and topics without making changes to the net-structure. This makes it possible to investigate different configuration of MQTT and it is a main benefit provided by CPNs in comparison to ordinary Petri Nets.

As described in [15], an MQTT client can operate as a publisher or subscriber. We use the term client to generally refer to a publisher or a subscriber. There is no restrictions in terms of hardware to run as a MQTT client, and any device equipped with a MQTT library and connected to a MQTT broker can operate as a client. The MQTT broker [15] is the core of any publish/subscribe protocol. Topics are the mechanism to filter messages in the MQTT protocol. A client can either subscribe to a topic to receive messages or publish on a topic. The broker is primarily responsible for receiving and filtering messages, deciding to which clients they will be dispatched and sending them to all subscribed clients. Another responsibility of the broker is client authentication and authorisation.

The two substitution transitions in Fig. 2 are connected via directed arcs to the two places CtoB and BtoC. The clients and the broker interact by producing and consuming tokens on the places. Figure 3 shows the central data type definitions used for the colour sets of the places CtoB and BtoC and the modelling of clients and messages. The colour sets QoS is used for modelling the three quality of service levels supported by MQTT, and the colour set PID is used for modelling the packet identifiers which plays a central role in the MQTT protocol logic. It can be seen that we have abstracted from the actual payload of



**Fig. 2.** The top-level module of the MQTT CPN model.

the published messages as these are not central for modelling the protocol logic. For similar reasons, we also abstract from the hierarchical structuring of topics.

The places **CtoB** and **BtoC** are designed to behave as queues. The purpose of this is to assure the ordered message distribution property specified in the MQTT documentation. Even so, these two places are slightly different; while **CtoB** is modelled as a single queue that the broker manages to consume messages, **BtoC** is designed to maintain an incoming queue of messages for each client. This construction assures that all clients will have their own queue, respecting the ordered reception of messages. The function `initMsgQueue()` initialises the queues according to the number of clients specified by the symbolic constant **C**. The **BrokerxMessages** colour set for the **CtoB** place used at the bottom of Fig. 3 consists of a list of **ClientxMessage** which are pairs of **Client** and **Messages**.

We represent all the messages that the clients and the broker can use by means of the **Message** colour set. We use the terms packet and message indistinguishably when we refer to control packets. The control information used depends on the messages considered. As an example, a **Connect** message (packet) does not contain control information, but a **Publish** message requires a specific **Topic**, **QoS**, and **PID**. The **Topic** and **QoS** colour sets are both indexed types containing values (`topic(1)`, `topic(2)` ... `topic(T)`) depending on the constant **T**, and `QoS(0)`, `QoS(1)` and `QoS(2)`, respectively. The **ClientxMessages** colour set for the **BtoC** place encapsulates all the queues (each one declared as a pair of **Client** and **Messages** in the **ClientxMessageQueue** colour set) in one single queue. This construction allows us to deal with the distribution of multiple messages in a single step in the broker side which in turn simplifies the modelling of the broker and reduces the number of reachable states of the model.

### 3 Modelling the Protocol Roles and their Interaction

We now consider the different phases and client-broker interaction in the MQTT protocol, and show how we have modelled the MQTT protocol logic using CPNs.

---

```

val T = 5; (* number of topics *)
val C = 2; (* number of clients *)

colset Client = index client with 1..C;
colset Topic  = index topic with 1..T;
colset QoS    = index QoS with 0..2; (* quality of service *)
colset PID    = INT;                (* packet identifiers *)

colset TopicxPID      = product Topic * PID;
colset TopicxQoSxPID = product Topic * QoS * PID;

colset Message = union CONNECT + CONNACK +
  SUBSCRIBE : TopicxQoSxPID + UNSUBSCRIBE : TopicxPID +
  SUBACK    : TopicxQoSxPID + UNSUBACK    : TopicxPID
  PUBLISH   : TopicxQoSxPID +
  PUBACK    : TopicxPID      + PUBREC      : TopicxPID +
  PUBREL    : TopicxPID      + PUBCOMP    : TopicxPID +
  DISCONNECT;

colset Messages = list Message;

colset ClientxMessage      = product Client * Message;
colset BrokerxMessages    = list ClientxMessage;

colset ClientxMessageQueue = product Client * Messages;
colset ClientxMessages     = list ClientxMessageQueue;

```

---

**Fig. 3.** Client and message colour set definitions

### 3.1 Interaction Overview

In order to show how clients and the broker interact, we describe the different actions that clients may carry by considering an example. Figure 4 shows a sequence diagram for a scenario where two clients connect, perform subscribe, publish and unsubscribe, and finally disconnect from the broker. The protocol interaction is as follows:

1. Client 1 and Client 2 request a connection to the Broker.
2. The Broker sends back a connection acknowledgement to confirm the establishment of the connection.
3. Client 2 subscribes to topic 1 with a QoS level 1, and the Broker confirms the subscription with a subscribe acknowledgement message.
4. Client 1 publishes on topic 1 with a QoS level 1. The Broker responds with a corresponding publish acknowledgement.
5. The Broker transmits the publish message to Client 2 which is subscribed to the topic.

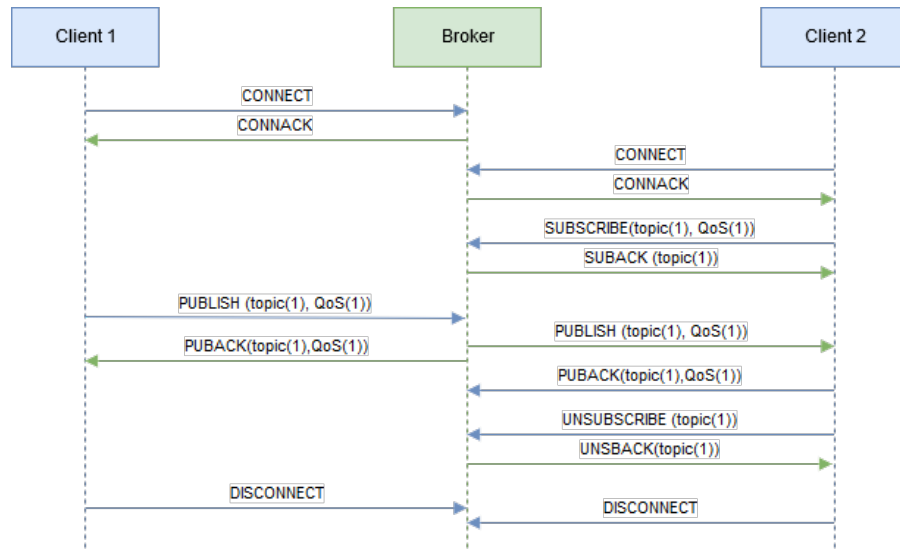


Fig. 4. Message sequence diagram illustrating the MQTT phases.

6. Client 2 gets the published message, and sends a publish acknowledgement back as a confirmation to the Broker that it has received the message.
7. Client 2 unsubscribes to topic 1, and the Broker responds with an unsubscribe acknowledgement.
8. Client 1 and Client 2 disconnect.

### 3.2 Client and Broker State Modelling

The colour sets defined for modelling the client state are shown in Fig. 5. The states of the clients are represented by the `ClientxState` colour set which is a product of `Client` and `ClientState`. The record colour set `ClientState` is used to represent the state of a client which consists of a list of `TopicxQoS`, a `State`, and a `PID`. Using this, a client stores the topics it is subscribed to, and the quality of service level of each subscription. The `State` colour set is an enumeration type containing the values `READY` (for the initial state), `WAIT` (when the client is waiting to be connected), `CON` (when the client is connected), and `DISC` (for when the client has disconnected).

Below we present selected parts of the model by first presenting a high-level view of the clients and broker sides, and then illustrating how the model captures the execution scenario described in Section 3.1 where two clients connect, one subscribes to a topic, and the other client publishes on this topic. The unsubscribe and the disconnection phases are not detailed due to space limitations.



---

```

colset State = with READY | DISC | CON | WAIT;

colset TopicxQoS      = product Topic * QoS;
colset ListTopicxQoS = list TopicxQoS;

colset ClientState   = record topics : ListTopicxQoS *
                        state : State *
                        pid    : PID;

colset ClientxState  = product Client * ClientState;
    
```

---

Fig. 5. Colour set definitions used for modelling client state.

### 3.3 Client Modelling

The ClientProcessing submodule in Fig. 6 models all the operations that a client can carry out. Clients can behave as senders and receivers, and the five substitution transitions CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE and DISCONNECT has been constructed to capture both behaviours.

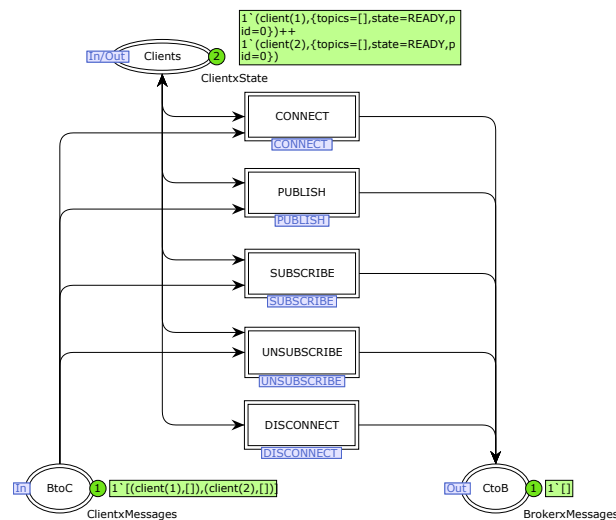


Fig. 6. ClientProcessing submodule.

The socket place Clients stores the information of all the clients that are created at the beginning of the execution of the model. In this scenario there are two clients, and the value of the tokens representing the state of the two

clients is provided in the green rectangle (the marking of the place) next to the Clients place. The BtoC and CtoB port places are associated with the socket places already shown in Fig. 3.

### 3.4 Broker Modelling

We have modelled the broker similarly as we have done for clients. This can be seen from Fig. 7 which shows the `BrokerProcessing` submodule. The `Connected-Clients` place keeps the information of all clients as perceived by the broker. This place is designed as a central storage, and it is used by the broker to distribute the messages over the network. The broker behaviour is different from that of the clients, since it will have to manage all the requests and generate responses for several clients at the same time.

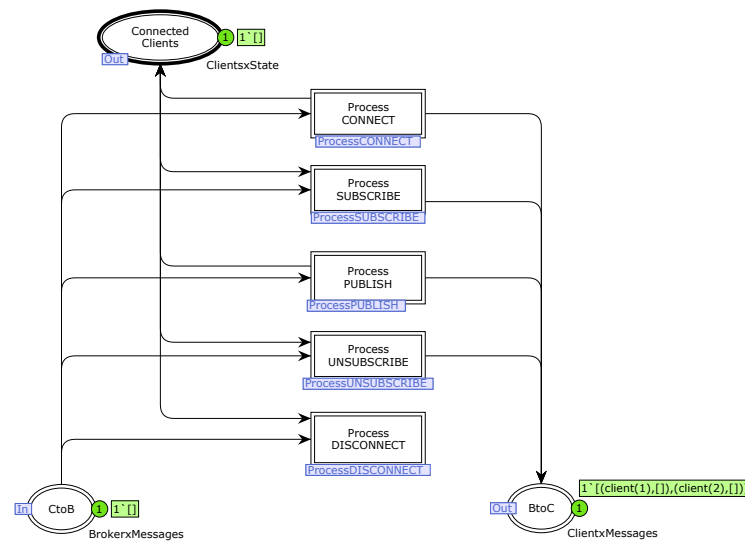


Fig. 7. The `BrokerProcessing` module.

### 3.5 Connection Phase

The first step for a client to be part of the message exchange is to connect to the broker. A client will send a `CONNECT` request, and the broker will respond with a `CONNACK` message to complete the connection establishment. Figure 8 shows the `CONNECT` submodule in a marking where `client(1)` has sent a `CONNECT` request and it is waiting (`state = WAIT`) for the broker acknowledgement processing to finish such that the connection state can be set to `CON`.

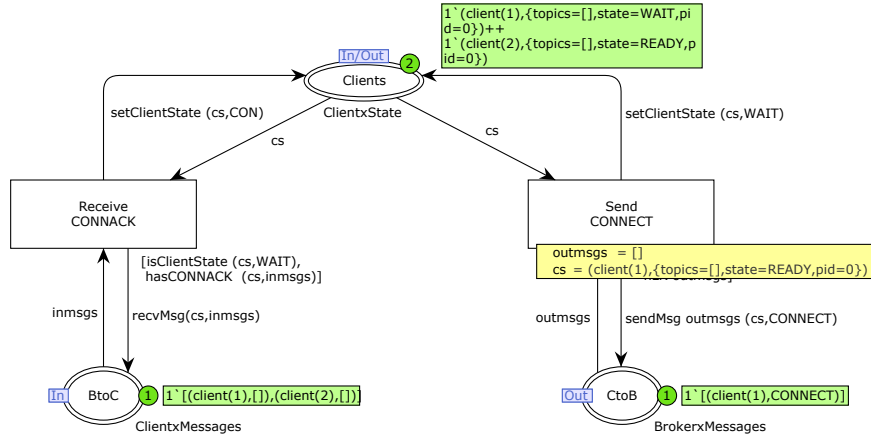


Fig. 8. CONNECT module after the send-connect occurrence.

The broker will receive the CONNECT request. The broker will register the client in the place ConnectedClients and send back the acknowledgement. Figure 9 shows the situation where *client(1)* is connected in the broker side and the CONNACK response has been sent back to the client. The function `connectClient()` used on the arc from the `ProcessCONNECT` transition to the `ConnectedClients` place will record the connected client on the broker side. The last step of the connection establishment will occur again in the clients side, where the transition `ReceiveCONNACK` (in Fig. 8) will be enabled, meaning that the confirmation for the connection of *client(1)* can proceed.

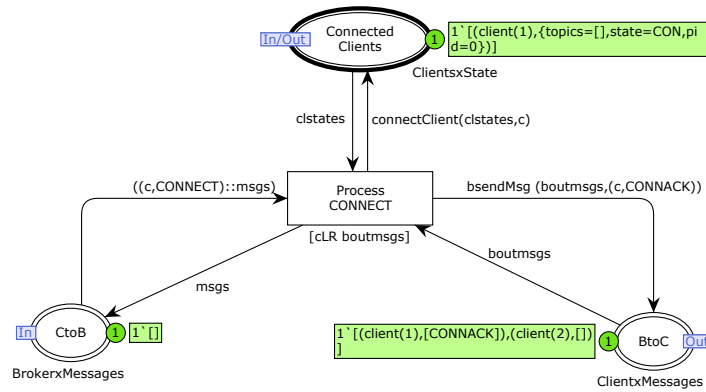


Fig. 9. ProcessCONNECT module after the process-connect occurrence.

### 3.6 Subscription Phase

Starting from the point where both clients are connected, i.e., for both clients, the state is `CON` as shown at the top of Fig. 10), the `client(2)` will send a `SUBSCRIBE` request to `topic(1)` with `QoS(1)`. The place `PendingAcks` represents a queue that each client maintains to store the `PIDs` that are waiting to be acknowledged. In this example, the message has assigned a `PID = 0`, and `client(2)` is waiting for an acknowledgement to this subscription with a `PID = 0`. When a client receives a `SUBACK` (subscribe acknowledgement) it will check that the packet identifier (0 in this case) is the same to ensure that the correct packet is being received. At the bottom right side of the Fig. 10, the message has been sent to the broker.

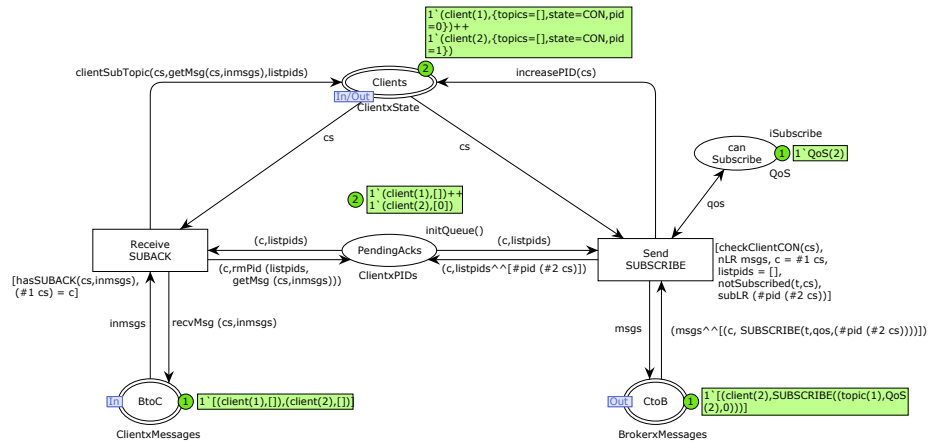


Fig. 10. SUBSCRIBE module after the subscribe occurrence.

We show now the situation where the `SUBSCRIBE` request has been processed by the broker as represented in Fig. 11. The function `brokerSubscribeUpdate()` manages the subscription process, so if the client is subscribing to a new topic, it will be added to the client state stored in the broker. If the client is already subscribed to this topic it will update it. In the example, one can see that `client(1)` keeps the same state, but `client(2)` has appended this new topic to its list. The corresponding `SUBACK` message has been sent to `client(2)` (with the `PID` set to 0) to confirm the subscription. Next, `client(2)` will detect that the response has arrived and it will check that the packet identifiers correspond to each other.

### 3.7 Publishing Phase

The publishing process in the considered scenario requires two steps to be completed. First a client sends a `PUBLISH` in a specific topic, with a specific `QoS`, which is received by broker. The broker will answer back with the corresponding

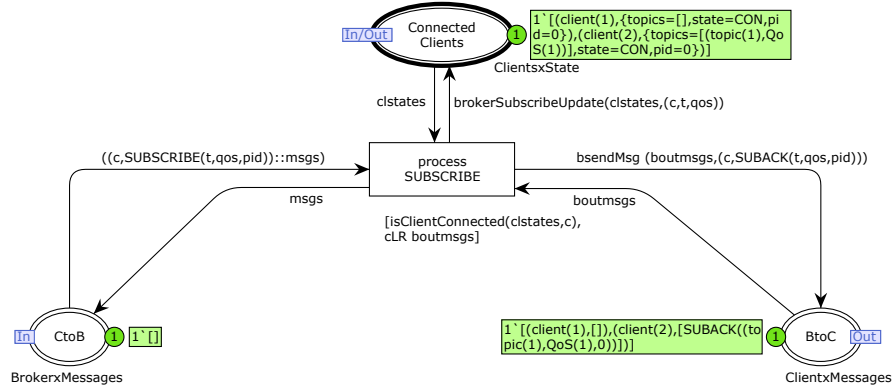


Fig. 11. ProcessSUBSCRIBE module after occurrence of ProcessSUBSCRIBE.

acknowledgement, depending on the quality of service previously set. Second, the broker, that stores information for all clients, will propagate the PUBLISH sent by the client to any clients subscribing to that topic. We have modelled the clients and broker sides using different submodules depending on the quality of service that is being applied for sending and receiving. In our example, `client(1)` will publish in `topic(1)` with a `QoS(1)`. This means that the broker will acknowledge back with a PUBLISH to `client(1)`, and will create a PUBLISH message for `client(2)`, which is subscribed to this topic with a `QoS(1)`. In this case, there is no downgrading for the `client(2)`, so the publication process will be similar to step 1, i.e. `client(2)` will send back a PUBLISH to the broker.

Figure 12 shows the scenario where `client(1)` has sent a PUBLISH with a `QoS(1)` for the `topic(1)`. Similar to the subscription process, the place `CtoB` holds the message that the broker will receive, and the place `Publishing` keeps the information (PID and `topic` in this case) of the packet that needs to be acknowledged. The transition `TimeOut` models the behaviour for the re-transmission of packets. Quality of service level 1 assures that the message will be received at least once. The `TimeOut` transition will be enabled to re-send the message until the client has received the acknowledgement from the broker.

The `Broker` module models the logic for both receiver and sender behaviours. Figure 13 shows a marking corresponding to the state where the broker has processed the PUBLISH request made by `client(1)`, and it has generated both the answer to this client and the PUBLISH message for `client(2)` (in this case, only one client is subscribed to the topic). The port place `BPID` (Broker PID), at top right of Fig. 13, will hold a packet identifier for each message that the broker re-publishes to the clients. The port place `Publishing` keeps information for all the clients that will acknowledge back the publish messages transmitted by the broker. Again, a `TimeOut` is modelled which, in this case, creates PUBLISH messages for all the clients subscribed to the topic in question. In the `BtoC` place (bottom right of Fig. 13), one can see that both messages have been sent, one



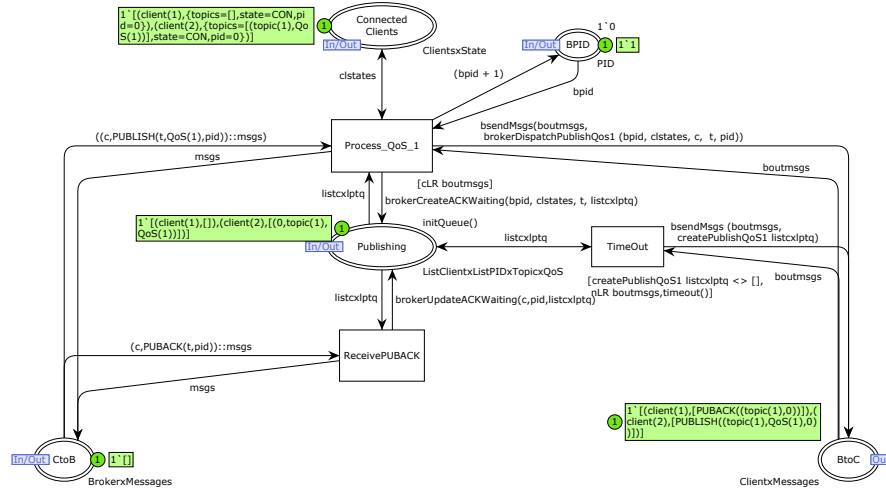


Fig. 13. Process\_QoS\_1 module after the Process\_QoS\_1 occurrence.

phenomenon. We identified several modelling errors in the course of conducting this incremental model validation based on the phases of the MQTT protocol.

To obtain a finite state space, we have to limit the number of clients and topics, and also bound the packet identifiers. It can be observed that there is no interaction between clients and brokers across topics as the protocol treats each topic in isolation. Executing the protocol with multiple topics is equivalent to running multiple instances of the protocol in parallel. We therefore only consider a single topic for the model validation. Initially, we consider two clients. The packet identifiers are incremented throughout the execution of the different phases of the protocol (connect, subscribe, data exchange, unsubscribe, and disconnect). This means that we cannot use a single global bound on the packet identifiers as a client could reach this bound, e.g., already during the publish phase and hence the global bound would prevent a subsequent unsubscribe to take place. We therefore introduce a local upper bound on packet identifiers for each phase. This local bound expresses that the given phase may use packet identifiers up to this local bound. Note that the use of bounds does not guarantee that the client uses packet identifiers up to bound. It is the guard on the transitions sending packets from the clients that ensures that these local bounds are accomplished. Finally, we enforce an upper bound on the number of messages that can be in the message queues on the places CtoB and BtoC.

Below we describe each step of the model validation and the behavioural properties verified. The properties verified in each step include the properties from the previous step. We summarise the experimental results at the end. For the actual checking of properties, we have used the state and action-oriented variant of CTL supported by the ASK-CTL library of CPN Tools.

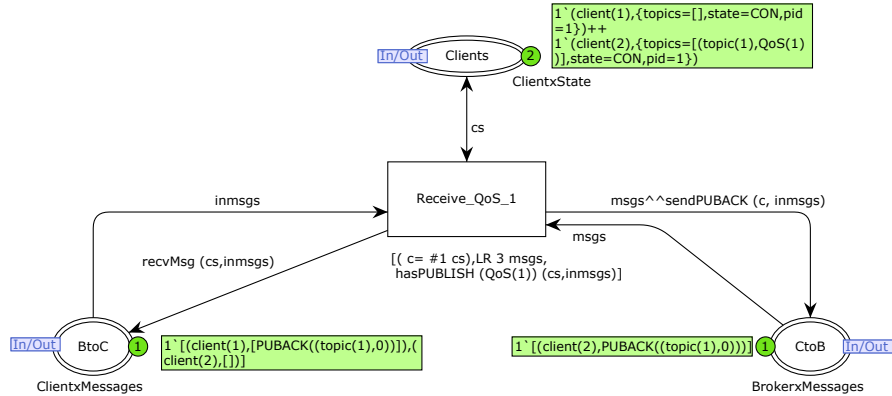


Fig. 14. Receive module after the transition `Receive_QoS_1` occurrence

**Step 1 – Connect and Disconnect.** In the first step, we consider only the part of the model related to clients connecting and disconnecting to the broker. The state space in this step is rather small, and consists of only 33 states, 44 arcs, and a single dead marking. We consider the following behavioural properties:

- S1-P1-ConsistentConnect** The clients and the broker have a consistent view of the connection state. This means that if the client in the clients side is in a connect state, then also the broker has the client recorded as connected.
- S1-P2-ClientsCanConnect** For each client, there exists a reachable state in which the client is connected to the broker.
- S1-P3-ConsistentTermination** In each terminal state (dead marking), clients are in a disconnect state, the broker has recorded the clients as disconnected, no clients are recorded as subscribed in both clients and broker sides, and there are no outstanding messages in the message buffers.
- S1-P4-PossibleTermination** The protocol can always be terminated, i.e., a terminal state (dead marking) can always be reached.

The two properties S1-P3 and S1-P4 imply partial correctness of the protocol as it states that the protocol can always be terminated, and if it terminates, then it will be in a correct state. The state space obtained in this step is acyclic which together with S1-P3 implies the stronger property of eventual correct termination. This is, however, more a property how the model has been constructed as in a real implementation there is nothing forcing a client to disconnect.

**Step 2 – Subscribe and Unsubscribe.** In the second step, we add the ability for the clients to subscribe and unsubscribe (in addition to connect and disconnect from step 1). The state space when adding these features has 1716 states, 4412 edges, and a single dead marking. For subscribe and unsubscribe we additionally consider the following properties:



- S2-P1-CanSubscribe** For each of the clients, there exists states in which both the clients and the broker sides consider the client to be subscribed.
- S2-P2-ConsistentSubscription** If the broker side considers the client to be subscribed, then the clients side considers the client to be subscribed.
- S2-P3-EventualSubscribed** If the client sends a subscribe message, then eventually both the clients and the broker sides will consider the client to be subscribed.
- S2-P4-CanUnsubscribe** For each client there exists executions in which the client sends an unsubscribe message.
- S2-P5-EventualUnsubscribed** If the client sends an unsubscribe message, then eventually both the clients and the broker sides considers the client to be unsubscribed.

It should be noted that for property S2-P2, the antecedent of the implication deliberately refers to the broker side. This is because the broker side unsubscribes the client upon reception of the unsubscribe message, whereas the client side does not remove the topic from the set of subscribed topics until the subscribe acknowledgement message is received from the broker. Hence, during unsubscribe, we may have the situation that the broker has unsubscribed the client, but the subscribe acknowledgement has not yet been received on the client side.

**Step 3 – Publish and QoS levels.** In this step we also consider publication of data for each of the three quality of service levels. As we do not model the concrete data contained in the messages, we use the packet identifiers attached to the message published to identity the packets being sent and received by the clients. In order to reduce the effect of state explosion, we verify properties for each QoS level in isolation. To make it simpler to check properties related to data being sent, we record for each client the packet identifiers of messages sent. For all three service levels, we consider the following properties:

- S3-P1-PublishConnect** A client only publishes a message if it is in a connected state.
- S3-P2-CanPublish** For each client there exists executions in which the client publishes a message.
- S3-P3-CanReceive** For each client there exists executions in which the client receives a message.
- S3-P4-Publish** Any data (packet identifiers) received on the client side must also have been sent on the client side.
- S3-P4-ReceiveSubscribed** A client only receives data if it is subscribed to the topic, i.e., the client side considers the client to be subscribed.

It should be noted that it is possible for a client to publish to a topic without being subscribed. The only requirement is that the client is connected to the broker. What data can correctly be received depends on the quality of service level considered. We therefore have one of the following three properties depending on the quality of service considered.

- S3-P5-Publish-QoS0** The data (packet identifiers) received by the subscribing clients must be a subset of the data (packet identifiers) sent by the clients.
- S3-P5-Publish-QoS1** The data sent on the client side must be a subset of the multi-set of packets received by the subscribing clients.
- S3-P5-Publish-QoS2** The data received by each client is identical to the packet identifiers sent by the clients.

To check the above properties related to data received, we accumulate the packet identifiers received such that they can be compared to the packet identifiers sent. To simplify the verification of data received, we force (using priorities) both clients to be subscribed before data exchange takes place since otherwise the data that can be received depends on the time at which the clients were subscribed and unsubscribed. As part of future work, we are investigating how to lift this restriction and express the data received properties more generally.

Table 1 summarises the verification results. We provide for each step information about the steps and configurations considered, the size of the state spaces, and the property verified. The DM column lists the number of dead markings. The state space generation and verification took in all cases less than one minute.

**Table 1.** Summary of configurations and experimental results for model validation

Step	Phases	State space			Behavioural Properties
		States	Arcs	DMs	
S1	Connect + Disconnect	33	44	1	ConsistentConnect ClientsCanConnect ConsistentTermination PossibleTermination
S2	Subscribe + Unsubscribe	1,716	4,412	1	Can{Subscribe,Unsubscribe} Eventual{Subscribe,Unsubscribe} ConsistentSubscription
S3	Publish +				PublishConnect,Can{Publish,Receive} Publish,ReceiveSubscribed
	Publish QoS 0	2,953	5,784	7	Publish-QoS0
	Publish QoS 1	8,329	15,606	7	Publish-QoS1
	Publish QoS 2	12,066	19,466	34	Publish-QoS2

## 5 Conclusions and Related Work

We have presented a formal CPN model based on the most recent specification of the MQTT protocol (version 3.1.1 [3]). The constructed CPN model represents a step towards developing a formal and executable specification of the MQTT protocol. For the validation of the model, we have conducted simulation and state space exploration in order to verify an extensive list of behavioural properties and thereby validate the correctness of the model. In addition to aiding in the development of inter-operable MQTT implementations, the CPN MQTT

model may also be used to perform model-based testing of MQTT protocol implementations following the approach presented in [17].

During our study of MQTT specification, we encountered several parts that are vaguely defined and which could lead developers to different implementations. There is a gap in the specification with the lossless property in that the MQTT protocol is described to run over TCP/IP, or over other transport protocols that provide ordered, lossless and bidirectional connections. However, the QoS level 0 establish that message loss can occurs and the specification is not clear as to whether this is related to termination of TCP connections and/or clients connecting and disconnecting to the broker. Additionally, we found several issues in the specification of QoS levels 1 and 2. As an example, it is specified that the receiver (assuming the broker role) is not required to complete delivery of the application message before sending the PUBACK (for QoS1) or PUBREC or PUBCOMP (for QoS2) and the decision of handling this is up to the implementer. Moreover, the documentation specifies that when the original sender receives the PUBACK packet, ownership of the application message is transferred to the receiver, but it is unclear how to determine that the original sender has received the PUBACK packet.

There exists previous work on modelling and validation of the MQTT protocol. In [11], the authors uses the UPPAAL SMC model checker [7] to evaluate different quantitative and qualitative (safety, liveness and reachability) properties against a formal model of the MQTT protocol defined with probabilistic timed automata. Compared to their work, we have verified a larger set of behavioural properties using the incremental approach adding more operations in each step. In [14], tests are conducted over three industrial implementations of MQTT against a subset of the requirements specified in the MQTT version 3.1.1 standard using the TTCN-3 test specification language. In comparison to our work, test-based approaches do not cover all the possible executions but only randomly generated scenarios. With the exploration of state spaces, we considered all the possible cases. In [2], the authors first define a formal model of MQTT based on timed message-passing process algebra, and they conduct analysis of the three QoS levels. In contrast, our work is not limited to the publishing/subscribing process, but considers all operations of the MQTT specification.

We see several directions for future work based on the result presented in this paper. Currently, we have restricted the model to simplify the verification part and reduce the effect of the state space explosion. We are currently investigating how to lift some of these restrictions to conduct verification of additional scenarios. Since our goal was to construct a complete model that covers all the MQTT operations, we have abstracted some parts, e.g., the topics structure and the payload of packets. We are considering how to incrementally add more details to the model so it will be even closer to the MQTT specification. Also, the model currently considers scenarios where the re-connection of clients is not taken into account. We plan to extend the model so that re-connection and simulation of persistence of data will be included, being able to also extend the behavioural properties verified.

## References

1. M. A. Adamski, A. Karatkevich, and M. Wegrzyn. *Design of embedded control systems*, volume 267. Springer, 2005.
2. B. Aziz. A formal model and analysis of an IoT protocol. *Ad Hoc Networks*, 36:49–57, 2016.
3. A. Banks and R. Gupta. MQTT Version 3.1.1. *OASIS standard*, 29, 2014. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>.
4. L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *Proceedings of the 29th international conference on Software Engineering*, pages 199–208. IEEE Computer Society, 2007.
5. J. Billington and M. Diaz. *Application of Petri nets to Communication Networks: Advances in Petri nets*, volume 1605. Springer Science & Business Media, 1999.
6. S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang. A vision of IoT: Applications, challenges, and opportunities with china perspective. *IEEE Internet of Things journal*, 1(4):349–359, 2014.
7. A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, Aug 2015.
8. J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3018 of *Lecture Notes in Computer Science*. Springer, 2004.
9. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.
10. D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Intl. SPIN Workshop on Model Checking of Software*, pages 166–180. Springer, 2003.
11. M. Houimli, L. Kahloul, and S. Benaoun. Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things. In *Mathematics and Information Technology*, pages 214–221. IEEE, 2017.
12. ITU-T. Overview of the Internet of things. <https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060>, 2012.
13. K. Jensen and L. Kristensen. Coloured Petri Nets: A Graphical Language for Modelling and Validation of Concurrent Systems. *Communications of the ACM*, 58(6):61–70, 2015.
14. K. Mladenov. *Formal verification of the implementation of the MQTT protocol in IoT devices*. Master thesis, University of Amsterdam, 2017.
15. MQTT essentials part 3: Client, broker and connection establishment. <https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe>.
16. A. Rodriguez, L. M. Kristensen, and A. Rutle. Complete CPN model of the MQTT Protocol. via Dropbox. <http://www.goo.gl/6FPVUq>.
17. R. Wang, L. Kristensen, H. Meling, and V. Stolz. Application of Model-based Testing on a Quorum-based Distributed Storage. In *Proc. of PNSE'17*, volume 1846 of *CEUR Workshop Proceedings*, pages 177–196, 2017.
18. F. Wortmann and K. Flüchter. Internet of things. *Business & Information Systems Engineering*, 57(3):221–224, 2015.
19. Y. Yoo, O. Henfridsson, and K. Lyytinen. Research commentary — the new organizing logic of digital innovation: an agenda for information systems research. *Information systems research*, 21(4):724–735, 2010.
20. L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/-subscribe architectures. In *Proc. of the SAVCBS*, volume 3, pages 35–41, 2003.